第8章

# NVIC与中断控制

* NVIC概览
* 中断配置基础
* 中断使能与除能
* 中断的悬起与解悬
* 中断建立全过程的演示
* 软件中断
* 再论SysTick定时器

## 8.1 NVIC概览

正如前文已经多次提到的，向量中断控制器，简称NVIC，是Cortex-M3不可分离的一部分，它与CM3内核的逻辑紧密耦合，有一部分甚至水乳交融在一起。NVIC与CM3内核同声相应，同气相求，相辅相成，里应外合，共同完成对中断的响应。NVIC的寄存器以存储器映射的方式来访问，除了包含控制寄存器和中断处理的控制逻辑之外，NVIC还包含了MPU、SysTick定时器以及调试控制相关的寄存器。本章中，我们将体检NVIC的中断处理控制逻辑。MPU与调试控制逻辑在后续章节中讨论。

NVIC共支持1至240个外部中断输入（通常外部中断写作IRQs）。具体的数值由芯片厂商在设计芯片时决定。此外，NVIC还支持一个“永垂不朽”的不可屏蔽中断（NMI）输入。NMI的实际功能亦由芯片制造商决定。在某些情况下，NMI无法由外部中断源控制。

NVIC的访问地址是0xE000\_E000。所有NVIC的中断控制/状态寄存器都只能在特权级下访问。不过有一个例外——软件触发中断寄存器可以在用户级下访问以产生软件中断。所有的中断控制／状态寄存器均可按字／半字／字节的方式访问。此外，还有几个中断掩蔽寄存器也与中断控制密切相关，它们是第三章中讲到的“特殊功能寄存器”，只能通过MRS/MSR及CPS来访问。

## 8.2 中断配置基础

每个外部中断都在NVIC的下列寄存器中“挂号”：

* + 使能与除能寄存器
  + 悬起与“解悬”寄存器
  + 优先级寄存器
  + 活动状态寄存器

另外，下列寄存器也对中断处理有重大影响

* + 异常掩蔽寄存器（PRIMASK, FAULTMASK以及BASEPRI）
  + 向量表偏移量寄存器
  + 软件触发中断寄存器
  + 优先级分组位段

## 8.3 中断的使能与除能

中断的使能与除能分别使用各自的寄存器来控制——这与传统的，使用单一比特的两个状态来表达使能与除能是不同的。CM3中可以有240对使能位／除能位(SETENA位/CLRENA位)，每个中断拥有一对。这240个对子分布在8对32位寄存器中（最后一对没有用完）。欲使能一个中断，我们需要写1到对应SETENA的位中；欲除能一个中断，你需要写1到对应的CLRENA位中。如果往它们中写0，则不会有任何效果。写零无效是个很关键的设计理念：通过这种方式，使能／除能中断时只需把“当事位”写成1，其它的位可以全部为零。再也不用像以前那样，害怕有些位被写入0而破坏其对应的中断设置（反正现在写0没有效果了），从而实现每个中断都可以自顾地设置，而互不侵犯——只需单一的写指令，不再需要读-改-写三步曲。

如上所述，SETENA位和CLRENA位可以有240对，对应的32位寄存器可以有8对，因此使用数字后缀来区分这些寄存器，如SETENA0, SETENA1…SETENA7，如表8.1所示。但是在特定的芯片中，只有该芯片实现的中断，其对应的位才有意义。因此，如果某个芯片支持32个中断，则只有SETENA0/CLRENA0才需要使用。SETENA/CLRENA可以按字/半字/字节的方式来访问。又因为前16个异常已经分配给系统异常，故而中断0的异常号是16，（回顾第7章中的表7.2）

表8.1 SETENA/CLRENA寄存器族 （此表参考官方技术参考手册作了些改编——译者注）

SETENAs: xE000\_E100 – 0xE000\_E11C ; CLRENAs:0xE000E180 - 0xE000\_E19C

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 类型 | 地址 | 复位值 | 描述 |
| SETENA0 | R/W | 0xE000\_E100 | 0 | 中断0-31的使能寄存器，共32个使能位  位[n]，中断#n使能（异常号16+n） |
| SETENA1 | R/W | 0xE000\_E104 | 0 | 中断32-63的使能寄存器，共32个使能位 |
| … | … | … | … | … |
| SETENA7 | R/W | 0xE000\_E11C | 0 | 中断224-239的使能寄存器，共16个使能位 |
|  |  |  |  |  |
|  |  |  |  |  |
| CLRENA0 | R/W | 0xE000\_E180 | 0 | 中断0-31的除能寄存器，共32个除能位  位[n]，中断#n除能（异常号16+n） |
| CLRENA1 | R/W | 0xE000\_E184 | 0 | 中断32-63的除能寄存器，共32个除能位 |
| … | … | … | … | … |
| CLRENA7 | R/W | 0xE000\_E19C | 0 | 中断224-239的除能寄存器，共16个除能位 |

## 8.4 中断的悬起与解悬

如果中断发生时，正在处理同级或高优先级异常，或者被掩蔽，则中断不能立即得到响应。此时中断被悬起。中断的悬起状态可以通过“中断设置悬起寄存器(SETPEND)”和“中断悬起清除寄存器(CLRPEND)”来读取，还可以写它们来手工悬起中断。

悬起寄存器和“解悬”寄存器也可以有8对，其用法和用量都与前面介绍的使能/除能寄存器完全相同，见表8.2。

表8.2 SETPEND/CLRPEND寄存器族 （此表参考官方技术参考手册作了些改编——译者注）

SETPENDs:0xE000\_E200 – 0xE000\_E21C ; CLRPENDs:0xE000E280 - 0xE000\_E29C

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 类型 | 地址 | 复位值 | 描述 |
| SETPEND0 | R/W | 0xE000\_E200 | 0 | 中断0-31的悬起寄存器，共32个悬起位  位[n]，中断#n悬起（异常号16+n） |
| SETPEND1 | R/W | 0xE000\_E204 | 0 | 中断32-63的悬起寄存器，共32个悬起位 |
| … | … | … | … | … |
| SETPEND7 | R/W | 0xE000\_E21C | 0 | 中断224-239的悬起寄存器，共16个悬起位 |
|  |  |  |  |  |
|  |  |  |  |  |
| CLRPEND0 | R/W | 0xE000\_E280 | 0 | 中断0-31的解悬寄存器，共32个解悬位  位[n]，中断#n解悬（异常号16+n） |
| CLRPEND1 | R/W | 0xE000\_E284 | 0 | 中断32-63的解悬寄存器，共32个解悬位 |
| … | … | … | … | … |
| CLRPEND7 | R/W | 0xE000\_E29C | 0 | 中断224-239的解悬寄存器，共16个解悬位 |

### 8.4.1 优先级

每个外部中断都有一个对应的优先级寄存器，每个寄存器占用8位，但是CM3允许在最“粗线条”的情况下，只使用最高3位。4个相临的优先级寄存器拼成一个32位寄存器。如前所述，根据优先级组的设置，优先级可以被分为高低两个位段，分别是抢占优先级和亚优先级。优先级寄存器都可以按字节访问，当然也可以按半字/字来访问。有意义的优先级寄存器数目由芯片厂商实现的中断数目决定，优先级配置寄存器的详细信息在附录D中给出（表D.18）。

表8.3 中断优先级寄存器阵列 0xE000\_E400 – 0xE000\_E4EF

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 类型 | 地址 | 复位值 | 描述 |
| PRI\_0 | R/W | 0xE000\_E400 | 0（8位） | 外中断#0的优先级 |
| PRI\_1 | R/W | 0xE000\_E401 | 0（8位） | 外中断#1的优先级 |
| … | … | … | … | … |
| PRI\_239 | R/W | 0xE000\_E4EF | 0（8位） | 外中断#239的优先级 |

表8.3B 系统异常优先级寄存器阵列 0xE000\_ED18 － 0xE000\_ED23

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 地址 | 名称 | 类型 | 复位值 | 描述 |
| 0xE000\_ED18 | PRI\_4 |  |  | 存储器管理fault的优先级 |
| 0xE000\_ED19 | PRI\_5 |  |  | 总线fault的优先级 |
| 0xE000\_ED1A | PRI\_6 |  |  | 用法fault的优先级 |
| 0xE000\_ED1B | - | - | - | - |
| 0xE000\_ED1C | - | - | - | - |
| 0xE000\_ED1D | - | - | - | - |
| 0xE000\_ED1E | - | - | - | - |
| 0xE000\_ED1F | PRI\_11 |  |  | SVC优先级 |
| 0xE000\_ED20 | PRI\_12 |  |  | 调试监视器的优先级 |
| 0xE000\_ED21 | - | - | - | - |
| 0xE000\_ED22 | PRI\_14 |  |  | PendSV的优先级 |
| 0xE000\_ED23 | PRI\_15 |  |  | SysTick的优先级 |

### 8.4.2 活动状态

每个外部中断都有一个活动状态位。在处理器执行了其ISR的第一条指令后，它的活动位就被置1，并且直到ISR返回时才硬件清零。由于支持嵌套，允许高优先级异常抢占某个ISR。然而，哪怕中断被抢占，其活动状态也依然为1（请仔细琢磨前文讲到的“直到ISR返回时才清零）。活动状态寄存器的定义，与前面讲的使能/除能和悬起/解悬寄存器相同，只是不再成对出现。它们也能按字／半字／字节访问，但他们是**只读**的，如表8.4所示。

表8.4 ACTIVE寄存器族 0xE000\_E300\_0xE000\_E31C（此表参考官方技术参考手册作了些改编——译者注）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 类型 | 地址 | 复位值 | 描述 |
| ACTIVE0 | RO | 0xE000\_E300 | 0 | 中断0-31的活动状态寄存器，共32个状态位  位[n]，中断#n活动状态（异常号16+n） |
| ACTIVE1 | RO | 0xE000\_E304 | 0 | 中断32-63的活动状态寄存器，共32个状态位 |
| … | … | … | … | … |
| ACTIVE7 | RO | 0xE000\_E31C | 0 | 中断224-239的活动状态寄存器，共16个状态位 |

### 8.4.3 特殊功能寄存器PRIMASK与FAULTMASK

PRIMASK用于除能在NMI和硬fault之外的所有异常，它有效地把当前优先级改为0（可编程优先级中的最高优先级）。该寄存器可以通过MRS和MSR以下例方式访问：

1. 关中断

MOV R0, #1

MSR PRIMASK, R0

2. 开中断

MOV R0, #0

MSR PRIMASK, R0

此外，还可以通过CPS指令快速完成上述功能：

CPSID i ;关中断

CPSIE i ;开中断

FAULTMASK更绝，它把当前优先级改为-1。这么一来，连硬fault都被掩蔽了。使用方案与PRIMASK的相似。但要注意的是，**FAULTMASK会在异常退出时自动清零**。

掩蔽寄存器虽然能一手遮天，却都动不了NMI，因为NMI是用在最危急的情况下的。因此系统为它开出单行道，无需挂号只是不要迟到。当NMI激活时，“谁都是省略号，唯独是你不得了，第一优先谁比你重要”！试想，如果NMI被连接到系统的掉电报警线上，且系统是体外循环机的电源管理器……如果因为中断被除能就视而不见，则会使体外循环机因断电而失能，体外循环序列可以被意外终止，病人的生命也将丢失。

### 8.4.4 BASEPRI寄存器

在更精巧的设计中，需要对中断掩蔽进行更细腻的控制——只掩蔽优先级低于某一阈值的中断——它们的优先级在数字上大于等于某个数。那么这个数存储在哪里？就存储在BASEPRI中。不过，如果往BASEPRI中写0，则另当别论——BASEPRI将停止掩蔽任何中断。例如，如果我们需要掩蔽所有优先级不高于0x60的中断，则可以如下编程：

MOV R0, #0x60

MSR BASEPRI, R0

如果需要取消BASEPRI对中断的掩蔽，则示例代码如下：

MOV R0, #0

MSR BASEPRI, R0

另外，我们还可以使用BASEPRI\_MAX这个名字来访问BASEPRI寄存器，它俩其实是同一个寄存器。但是当我们使用这个名字时，会使用一个条件写操作。个中原因如下：尽管它俩在硬件水平上是同一个寄存器，但是生成的机器码不一样，从而硬件的行为也不同：使用BASEPRI时，可以任意设置新的优先级阈值；但是使用BASEPRI\_MAX时则“许进不许出”——只允许新的优先级阈值比原来的那个在数值上更小，也就是说，只能一次次地扩大掩蔽范围，反之则不行。就好像绳子打了死结，只会越拉越紧。举例来说，检视下面的程序片断：

MOV R0, #0x60

MSR BASEPRI\_MAX, R0 ;掩蔽优先级不高于0x60的中断

MOV R0, **~~#0xf0~~**

MSR BASEPRI\_MAX, R0 ;本次设置被忽略，因为0xf0比0x60的优先级低

MOV R0, #0x40

MSR BASEPRI\_MAX, R0 ;Ok。扩大掩蔽范围到优先级不高于0x40的中断

为了把掩蔽阈值降低，或者解除掩蔽，需要使用“BASEPRI”这个名字。上例中，把设置阈值为0xf0的那条指令改用BASEPRI，则可以操作成功。显然，在用户级下是不得更改BASEPRI寄存器的。与其它和优先级有关的寄存器一样，系统中表达优先级的位数，也同样影响BASEPRI中有意义的位数。如果系统中只使用3个位来表达优先级，则BASEPRI有意义的值仅为0x00, 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0以及0xE0。

### 8.4.5 其它异常的配置寄存器

用法fault，总线fault以及存储器管理fault都是特殊的异常，因此给它们开了小灶。其中，它们的使能控制是通过“系统Handler控制及状态寄存器(SHCSR)”（地址：0xE000\_ED24）来实现的。各种faults的悬起状态和大多数系统异常的活动状态也都在该寄存器中，如表8.5所示。

表8.5 系统Handler控制及状态寄存器SHCSR（地址：0xE000\_ED24）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 18 | **USGFAULTENA** | R/W | 0 | 用法fault服务例程使能位 |
| 17 | **BUSFAULTENA** | R/W | 0 | 总线fault服务例程使能位 |
| 16 | **MEMFAULTENA** | R/W | 0 | 存储器管理fault服务例程使能位 |
| 15 | **SVCALLPENDED** | R/W | 0 | SVC悬起中。本来已经要SVC服务例程，但是却被更高优先级异常取代 |
| 14 | **BUSFAULTPENDED** | R/W | 0 | 总线fault悬起中，细节同上。 |
| 13 | **MEMFAULTPENDED** | R/W | 0 | 存储器管理fault悬起中，细节同上 |
| 12 | **USGFAULTPENDED** | R/W | 0 | 用法fault悬起中，细节同上 |
| 11 | **SYSTICKACT** | R/W | 0 | SysTick异常活动中 |
| 10 | **PENDSVACT** | R/W | 0 | PendSV异常活动中 |
| 9 | **-** | - | - | - |
| 8 | **MONITORACT** | R/W | 0 | Monitor异常活动中 |
| 7 | **SVCALLACT** | R/W | 0 | SVC异常活动中 |
| 6:4 | **-** | - | - | - |
| 3 | **USGFAULTACT** | R/W | 0 | 用法fault异常活动中 |
| 2 | **-** | - | - | - |
| 1 | **BUSFAULTACT** | R/W | 0 | 总线fault异常活动中 |
| 0 | **MEMFAULTACT** | R/W | 0 | 存储器管理fault异常活动中 |

写这些寄存器时要小心，必须确保对活动位的修改是经过深思熟虑的，决不能粗心修改。否则，如果某个异常的活动位被意外地清零了，其服务例程却不知晓，仍然执行异常返回指令，那么CM3将视之为无理取闹——在异常服务例程以外做异常返回，从而产生一个fault。

译注：下段文字改编自《Cortex-M3 Technical Reference Manual》, pg8-29，是给那些骨灰级玩家们看的，因为修改这些位还有更深层次的背景和特效。译文为：上表中的活动位虽然也是可写的，但是改动时必须予以极度的小心，否则这是玩火行为——设置或者清零这些位，会改变处理器中对异常活动的记录，却不会对应地修复堆栈中的数据（不会为了此改动而特意执行一次自动入栈或自动出栈操作），于是埋下了破坏堆栈内容而引起程序跑飞的隐患；另外，其它一些重要的数据结构也得不到清除，后患无穷。事实上，只有操作系统在特殊场合下才会修改它们。例如：在任务执行系统调用的过程中执行上下文切换（大幅提升实时性），或者在使用软件模拟未定义指令的功能期间（在用法fault服务例程中），以及软件模拟协处理器的功能期间，执行上下文切换，同样大幅提升实时性。

下面开始讲中断控制及状态寄存器ICSR。对于NMI、SysTick定时器以及PendSV，可以通过此寄存器手工悬起它们。另外，在该寄存器中，有好多位段都用于调试目的。在大多数情况下，它们对于应用软件都没有什么用处，只有悬起位对应用程序常常比较有参考价值，如表8.6所示。

表8.5 中断控制及状态寄存器ICSR（地址：0xE000\_ED04）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 31 | NMIPENDSET | R/W | 0 | 写1以悬起NMI。因为NMI的优先级最高且从不掩蔽，在置位此位后将立即进入NMI服务例程。 |
| 28 | PENDSVSET | R/W | 0 | 写1以悬起PendSV。读取它则返回PendSV的状态 |
| 27 | PENDSVCLR | W | 0 | 写1以清除PendSV悬起状态 |
| 26 | PENDSTSET | R/W | 0 | 写1以悬起SysTick。读取它则返回PendSV的状态 |
| 25 | PENDSTCLR | W | 0 | 写1以清除SysTick悬起状态 |
| 23 | ISRPREEMPT | R | 0 | =1时，则表示一个悬起的中断将在下一步时进入活动状态（用于单步执行时的调试目的） |
| 22 | ISRPENDING | R | 0 | 1=当前正有外部中断被悬起（不包括NMI） |
| 21:12 | VECTPENDING | R | 0 | 悬起的ISR的编号。如果不止一个中断悬起，则它的值是这次中断中，优先级最高的那一个。 |
| 11 | RETTOBASE | R | 0 | 如果异常返回后将回到基级(base level)，并且没有其它异常悬起时，此位为1。若是在线程模式下，在某个服务例程中，有不止一级的异常处于活动状态，或者在异常没有活动时执行了异常服务例程（此时执行返回指令将产生fault。此乃高危行为，大虾也需慎用），则此位为0 |
| 9:0 | VECTACTIVE | R | 0 | 当前活动的ISR编号，该位段指出当前运行中的ISR是哪个中断的（提供异常序号），包括NMI和硬fault。如果多个异常共享一个服务例程，该例程可根据本位段的值来判定是哪一个异常的响应导致它的执行。把本位段的值减去16,就得到了外中断的编号，并可以用此编号来操作外中断相关的使能/除能等寄存器。 |

## 8.5 中断系统设置全过程的演示

下面给出一个简单的例子，以演示如何建立一个外部中断。

1. 当系统启动后，先设置优先级组寄存器。缺省情况下使用组0（7位抢占优先级，1位亚优先级）。

2. 如果需要重定位向量表，先把硬fault和NMI服务例程的入口地址写到新表项所在的地址中。

3. 配置向量表偏移量寄存器，使之指向新的向量表（如果有重定位的话）

4. 为该中断建立中断向量。因为向量表可能已经重定位了，保险起见需要先读取向量表偏移量寄存器的值，再根据该中断在表中的位置，计算出对应的表项，再把服务例程的入口地址填写进去。如果一直使用ROM中的向量表，则无需此步骤。

5. 为该中断设置优先级。

6. 使能该中断

示例汇编代码如下：

LDR R0, =0xE000ED0C ; 应用程序中断及复位控制寄存器

LDR R1, =0x05FA0500 ; 使用优先级组5 (2/6)

STR R1, [R0] ; 设置优先级组

...

MOV R4, #8 ; ROM中的向量表

LDR R5, =(NEW\_VECT\_TABLE+8)

LDMIA R4!, {R0-R1} ; 读取NMI和硬fault的向量

STMIA R5!, {R0-R1} ; 拷贝它们的向量到新表中

...

LDR R0, =0xE000ED08 ; 向量表偏移量寄存器的地址

LDR R1, =NEW\_VECT\_TABLE

STR R1, [R0] ; 把向量表重定位

...

LDR R0, =IRQ7\_Handler ; 取得IRQ #7服务例程的入口地址

LDR R1, =0xE000ED08 ; 向量表偏移量寄存器的地址

LDR R1, [R1]

ADD R1, R1,#(4\*(7+16)); 计算IRQ #7服务例程的入口地址

STR R0, [R1] ; 在向量表中写入IRQ #7服务例程的入口地址

...

LDR R0, =0xE000E400 ; 外部中断优先级寄存器阵列的基地址

MOV R1, #0xC0

STRB R1, [R0,#7] ; 把IRQ #7的优先级设置为0xC0

...

LDR R0, =0xE000E100 ; SETEN寄存器的地址

MOV R1, #(1<<7) ; 置位IRQ #7的使能位

STR R1, [R0] ; 使能IRQ #7

另外，如果优先级组的设置使得中断嵌套层次可以很深，则务请确认主堆栈的容量足够用。因为异常服务程序总是使用MSP，为安全起见，主堆栈的容量应是最大可能需求的量（嵌套最深时需要的量）。

如果应用程序储存在ROM中，并且不需要改变异常服务程序，则我们可以把整个向量表编码到ROM的起始区域（从0地址开始的那段）。在这种情况下，向量表的偏移量将一直为0,并且中断向量一直在ROM中，因此上例可以大大简化，只需3步：

1. 建立优先级组

2. 为该中断指定优先级

3. 使能该中断

如果在I/O密集型系统中，软件需要控制大量的硬件设备，则可能必须要考虑如下因素：

* 该芯片支持的中断数
* 该芯片中表达优先级的位数

在CM3的NVIC中，有一个名为“中断控制器类型寄存器”，它提供了该芯片中支持的中断数目，粒度是32的整数倍，（如表8.7所示）。如果你嫌它太粗枝大叶，也可以通过对每个SETENA位进行先写后读的测试，来获取支持的中断的精确数目（往各SETENA中写1，不支持的中断将永远读回0，求出第1个0的位置即可），亦可使用SETPEND等其它位来做此测试。这主要用于需要适应不同芯片的程序。如果已经确定使用固定的芯片，则无需多此一举。

表8.7 中断控制器类型寄存器ICTR（地址：0xE000\_E004）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 4:0 | INTLINESUM | R | - | 中断输入的数量，以32为粒度，如  0=1至32  1=33至64  2=65至96  … |

为了判定正在使用的芯片使用了多少位来表达优先级，也可使用类似的方法：往某个优先级寄存器中写入0xFF，再读回来。则从MSB开始，有多少位是1就有多少位表达优先级。最少要使用3个位，此时你读回的是0xE0。

## 8.6 软件中断

软件中断，包括手工产生的普通中断，能以多种方式产生。最简单的就是使用相应的SETPEND寄存器；而更专业更快捷的作法，则是通过使用软件触发中断寄存器STIR，如表8.8所示。

表8.8 软件触发中断寄存器STIR（地址：0xE000\_EF00）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 8:0 | INTID | W | - | 影响编号为INTID的外部中断，其悬起位被置位。例如，写入8，则悬起IRQ #8 |

注意：系统异常（NMI，faults，PendSV等），不能用此法悬起。而且缺省时根本不允许用户程序改动NVIC寄存器的值。如果确实需要，必须先在NVIC的配置和控制寄存器(0xE000\_ED14)中，把比特1（USERSETMPEND）置位，才能允许用户级下访问NVIC的STIR。

## 8.7 SysTick定时器

SysTick定时器被捆绑在NVIC中，用于产生SysTick异常（异常号：15）。在以前，操作系统还有所有使用了时基的系统，都必须一个硬件定时器来产生需要的“滴答”中断，作为整个系统的时基。滴答中断对操作系统尤其重要。例如，操作系统可以为多个任务许以不同数目的时间片，确保没有一个任务能霸占系统；或者把每个定时器周期的某个时间范围赐予特定的任务等，还有操作系统提供的各种定时功能，都与这个滴答定时器有关。因此，需要一个定时器来产生周期性的中断，而且最好还让用户程序不能随意访问它的寄存器，以维持操作系统“心跳”的节律。

Cortex-M3处理器内部包含了一个简单的定时器。因为所有的CM3芯片都带有这个定时器，软件在不同 CM3器件间的移植工作就得以化简。该定时器的时钟源可以是内部时钟（FCLK，CM3上的自由运行时钟），或者是外部时钟（ CM3处理器上的STCLK信号）。不过，STCLK的具体来源则由芯片设计者决定，因此不同产品之间的时钟频率可能会大不相同。因此，需要检视芯片的器件手册来决定选择什么作为时钟源。

SysTick定时器能产生中断，CM3为它专门开出一个异常类型，并且在向量表中有它的一席之地。它使操作系统和其它系统软件在CM3器件间的移植变得简单多了，因为在所有CM3产品间，SysTick的处理方式都是相同的。

有4个寄存器控制SysTick定时器，如表8.9至表8.12所示。

表8.9 SysTick控制及状态寄存器（地址：0xE000\_E010）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 16 | COUNTFLAG | R | 0 | 如果在上次读取本寄存器后，SysTick已经计到了0，则该位为1。如果读取该位，该位将自动清零 |
| 2 | CLKSOURCE | R/W | 0 | 0=外部时钟源(STCLK)  1=内核时钟(FCLK) |
| 1 | TICKINT | R/W | 0 | 1=SysTick倒数计数到0时产生SysTick异常请求  0=数到0时无动作 |
| 0 | ENABLE | R/W | 0 | SysTick定时器的使能位 |

表8.10 SysTick重装载数值寄存器（地址：0xE000\_E014）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 23:0 | RELOAD | R/W | 0 | 当倒数计数至零时，将被重装载的值 |

表8.11 SysTick当前数值寄存器（地址：0xE000\_E018）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 23:0 | CURRENT | R/Wc | 0 | 读取时返回当前倒计数的值，写它则使之清零，同时还会清除在SysTick控制及状态寄存器中的COUNTFLAG标志 |

表8.10 SysTick校准数值寄存器（地址：0xE000\_E01C）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 31 | NOREF | R | - | 1=没有外部参考时钟（STCLK不可用）  0=外部参考时钟可用 |
| 30 | SKEW | R | - | 1=校准值不是准确的10ms  0=校准值是准确的10ms |
| 23:0 | TENMS | R/W | 0 | 在10ms的间隔中倒计数的格数。芯片设计者应该通过Cortex-M3的输入信号提供该数值。若该值读回零，则表示无法使用校准功能 |

校准值寄存器提供了这样一个解决方案：它使系统即使在不同的CM3产品上运行，也能产生恒定的SysTick中断频率。最简单的作法就是：直接把TENMS的值写入重装载寄存器，这样一来，只要没突破系统的“弹性极限”，就能做到每10ms来一次 SysTick异常。如果需要其它的SysTick异常周期，则可以根据TENMS的值加以比例计算。只不过，在少数情况下，CM3芯片可能无法准确地提供TENMS的值（如，CM3的校准输入信号被拉低），所以为保险起见，最好在使用TENMS前检查器件的参考手册。

SysTick定时器除了能服务于操作系统之外，还能用于其它目的：如作为一个闹铃，用于测量时间等。要注意的是，当处理器在调试期间被喊停（halt）时，则SysTick定时器亦将暂停运作。